The Amoeba interaction network monitor—initial results

Paul Ashton
Department of Computer Science
University of Canterbury

TR-COSC 09/95, Oct 1995

The Amoeba interaction network—initial results

Paul Ashton*

Department of Computer Science
University of Canterbury

TR COSC 09/95, October 1995

1 Introduction

The interaction network has been proposed as a way of representing all processing (process execution and message passing) that is the direct result of a single user input to any type of computer system [2, 3]. The interaction network is particularly well suited for representing interactive processing in distributed systems. The first interaction network monitor was developed for the SunOS operating system. Some interesting results were obtained using the monitor, but by the time it was completed the Sun 3 hardware and SunOS 4.0 software were basically obsolete.

A second interaction network monitor has been developed for the Amoeba distributed operating system [5, 6]. Amoeba was seen as a better development environment than SunOS for two main reasons. Firstly, Amoeba was designed from scratch as a distributed operating system. Secondly, the source code of Amoeba is readily available.

The main aim of this report is to document initial experiences with the Amoeba interaction network monitor. It begins with a very brief review of the interaction network concept, before describing the current implementation of the Amoeba interaction network monitor. Then some early experiences with use of the monitor are reported. The interaction networks presented do not constitute a comprehensive study of Amoeba performance, but do provide some interesting insights into the operation of Amoeba as well as showing the promise of the interaction network approach.

Note: the figures in this document are in colour, and are best viewed in colour. The figures can still be understood if the document is viewed in monochrome, but viewing in colour is definitely preferable.

2 The interaction network

Each interaction network represents the system reaction to a single user action (such as a key-stroke or mouse event). A system reaction is quite different to the execution of a single program because the system reaction involves complete and partial execution of many

*e-mail: paul@cosc.canterbury.ac.nz

programs. We regard reactions as being performed by communicating threads that are distributed over one or more nodes.

An interaction network is an acyclic digraph. Each vertex in the graph represents an event and each edge represents a period of thread execution, or message progression, between two events. To date, we have developed two ways of analysing interaction networks. One analysis method involves producing reports that contain the times spent in various states during the system reaction, and counts of activities of various types. The other analysis method is to provide a browser that produces a display of an interaction network.

The interaction network concept was developed originally to allow analysis of interactive performance, and the reports on resource-use produced by the first analysis method contain information intended for use in performance analysis. We have, however, found interaction networks to be useful in **any** situation in which increased understanding of interactive processing is required, with interaction network displays being especially useful. An interaction browser is the main analysis tool developed to date for Amoeba interaction networks (as will become apparent from the case studies). A program that produces a textual dump of each event record in a log file has been used in calculation of some of the totals reported in the case studies.

3 Overview of the Amoeba monitor

The design and implementation of the Amoeba interaction network monitor draws heavily on the work done on the SunOS monitor. A set of probes has been added to the Amoeba kernel to record the occurrence of various events, and to maintain the identifiers used to group event records into interaction networks. Each probe invokes a local event recorder to have an event record created, and ultimately stored to disk. Analysis tools extract interaction networks from a set of related log files, and provide textual and graphical views of interaction networks.

3.1 Probes added

To date a small set of probes has been added to the Amoeba kernel. Most of the current probes record "structural events"—the events that must be recorded to show the extent of a system reaction, where the extent of a reaction is the messages and periods of thread execution that resulted from a given user input. Probes for recording resource (CPU, disk, network, and so on) use have yet to be added.

The probes added to date can be classified into three groups:

1. Detection of user input. One probe has been added to the terminal server to record whenever a user input occurs on a serial line. Console keyboard and mouse input have yet to be instrumented, so all interaction networks presented here resulted from keyboard input through a character terminal interface.

2. Thread life-cycle[1]. Probes have been added to detect thread creation and termination, and when a process is assigned a new name.

[1] In Amoeba, a process is basically an address space. The kernel supports multiple threads of execution within each process. In the remainder of this document, the Amoeba definitions for the terms "process" and "thread" will be used.

3. Communication. Most communication in Amoeba is done using remote procedure calls. Probes have been added to record the sending and receiving of remote procedure call and return messages. Probes have also been added to the FLIP layer, the protocol layer below the RPC layer. These probes record sending and receiving of FLIP messages. The FLIP probes were added to better support clock synchronisation, a subject discussed in Section 3.3.

In addition to making a call to the event recorder, each probe is responsible for maintaining identifiers used to associate event records with system reactions. The techniques developed for this purpose in the SunOS interaction network monitor (described in detail in [1]) have been used successfully in the Amoeba monitor. There was one situation in which non-standard code was needed to propagate interaction identifiers. This case is discussed further in Section 10 below.

3.2 The event recorder

During monitoring, event records are logged to disk by an event recorder that is divided into client and server components. The servers, one in each kernel, assemble event records into buffers. Each probe logs event records to the server in its local kernel. Clients retrieve the buffers and write them to disk.

At boot time, 2 threads are created within the kernel to carry out calls made to the event recorder server. The following calls are provided by the server:

- calls to enable and disable event recording.
- a call to retrieve a buffer containing event records.
- a call to add an event record to a buffer. This is intended for probes outside the kernel, with probes inside the kernel using a direct function call to record an event record.

When the event recording client runs, it first calls the server to enable monitoring, then goes into a loop in which a call is made to the server to retrieve a buffer full of event records, and the returned buffer is logged to disk. A double buffering scheme is used for storing event records within the server. Each buffer is 30000 bytes. The RPC to get a buffer of event records does not return until a full buffer is available.

Because the server has an RPC interface, the client need not run on the same machine as the server. In fact, in all of the case studies reported here, all clients were executed on a host running SunOS so as to minimise the monitoring overhead on the Amoeba system.

3.3 Clock synchronisation

A Sparcstation 1 has two hardware timers with microsecond resolution. One is used by the kernel to provide standard timing services. The second has been taken over for use by the event recorder. The interrupt frequency of this second timer is very low (one interrupt every two seconds), so the chance of losing an interrupt is extremely small.

No attempt is made to synchronise clocks during event recording. Once monitoring is complete, there is one file of event records for each node that was monitored. A clock correction program has been written that takes two files of event records and corrects the timestamps in one to be consistent with the timestamps in the other. Several algorithms have been

implemented, including those described by Duda *et. al.* [4], enhanced versions of Duda's algorithms, and some new algorithms. Research into the effectiveness of the various algorithms is continuing, with results to date showing that very good synchronisation can be achieved.

3.4 Analysis tools

The following analysis tools have been developed:

1. `insplit`, a program that takes a collection of log files recorded at the same time on a collection of nodes, and from them produces a collection of log files each of which contains the event records of a single interaction network.

2. `logdump`, a program that produces a textual description of every event record in a log file.

3. `totcl` and `browser`, that together provide a graphical browser for display and inspection of an interaction network. `totcl` takes an interaction network log file and produces an intermediate text file that the TCL/Tk program `browser` can read.

Most analysis tools are adaptations of code from the SunOS monitor [1]. `browser` was written from scratch, but could easily be adapted for use with the SunOS monitor.

4 Introduction to the case studies

The case studies are illustrated with displays produced by the browser. Before presenting the case studies, we give some information on the display format used, and on the configuration of the Amoeba system on which the case studies were performed.

In an interaction network display, the vertices (each representing an event) are arranged in columns, with one column for each thread that performed processing in response to the user input. Time increases as you go down the network, with the Y-coordinate of each vertex in direct proportion to the time at which the event represented by the vertex occurred. The length of an edge in the Y direction shows, therefore, the duration of the activity that it represents. All of the vertical edges represent activities carried out by threads. Each non-vertical edge represents a message, or the initial activity performed by a newly-created thread.

The Amoeba system used in the case studies consisted of five Sparcstation 1 machines connected by an ethernet segment that was bridged off the department's network. All five machines ran Amoeba 5.2, with all patches up to patch 5 applied. One machine, scooter, was the file server, which ran soap (the directory server) and bullet (the file server). All five machines were considered by the run server whenever it was asked to select a machine on which a new process should be started.

The browser uses colour to highlight execution on different machines. Each machine has its own colour: green for scooter, light blue for piggy, red for ralph, purple for grover and grey for gonzo. All vertices and edges associated with a thread have the colour of the machine on which the thread ran. Vertices and edges associated with messages local to a machine have the colour of that machine. Vertices and edges associated with messages that were sent across the network are coloured black.

5 Case 1—Finding a bug in the MMU context flushing code

Early in the development of the monitor, there were problems with timestamping of event records. Clock ticks were being lost even though the interrupt frequency (at that stage) was one interrupt per second. The most likely explanation was that interrupts were being disabled for periods in excess of one second. The interaction network in Figure 1 was recorded before this problem had been resolved. The user action was to type newline at the end of a simple shell command line (in this case the command executed by the shell was the `date` command).

Two lengthy periods of inactivity can be seen from the interaction network. Both occur just after creation of a new process. It seemed likely that these periods (each of about 1.08 seconds, out of a total response time of 2.8 seconds) were periods during which interrupts were disabled. Addition of new probes and investigation of the source code to determine code executed with interrupts disabled helped to track down the problem. There was a bug in the code for invalidating an MMU context, that meant that many more addresses than necessary were accessed during context invalidation. The bug was duly fixed, and the problem of lost clock ticks disappeared. Flushing an MMU context now takes a little under 4ms.

6 Case 2—`ls`

The interaction network in Figure 2 shows execution of the `ls` command. The user action was entry of the newline character at the end of a command line on which the user had previously entered `l` and `s`. A total of 11 threads, executing in 8 different processes spread over 4 machines were involved in the system reaction. The system reaction took 0.569 seconds. The reaction involved 36 RPCs, and 191 ethernet packets containing a total of 182.7Kb of data. The ethernet utilisation averaged over the period of the reaction was, therefore, 26.3%.

In the figure, each thread is identified by machine (given by the colour of the thread's vertices) and the two numbers at the top the thread's column. The first number is a process id, and the second a thread id. Both are unique within a node. The kernel of an Amoeba machine executes as process 0. A summary of threads involved in the reaction, in the order in which they appear in the figure, appears in Table 1.

The source event of the interaction network is in the top left corner of the figure. The message from the terminal server to the shell contains the newline character, and is a reply to an RPC request that the shell had made earlier. The shell then consults the directory server to find the capability for the executable file to run. The session server, involved in implementing Unix emulation, is also accessed during this period. The shell accesses the file server, probably to read the header of the executable to get details of the type of the executable file. The run server is then asked to select a machine on which the new command will be run, and it selects grover.

The shell then calls the process/segment server on grover to have it create the new process. This remote procedure call takes 0.253 seconds, 45% of the total response time. The process/segment server spends most of its time mapping three segments into the address space of the new process, as in Amoeba the entire address space of a process must be loaded into physical memory before a process can run. The first four RPCs to the file server load the first segment. At 96044 bytes, this segment is most likely the `ls` executable. A second segment of 19284 bytes is then loaded from the file server. The third segment, of 64Kb, is loaded from the process/segment server on ralph (the machine on which the creating process

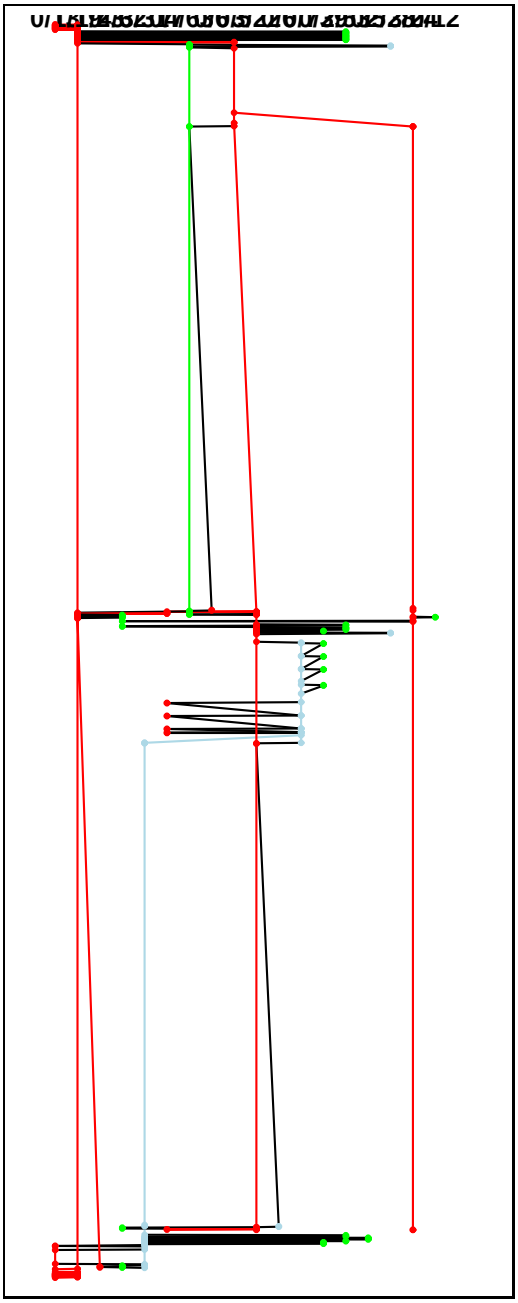

Figure 1: Interaction network showing execution of a small program before the context flushing bug was fixed.

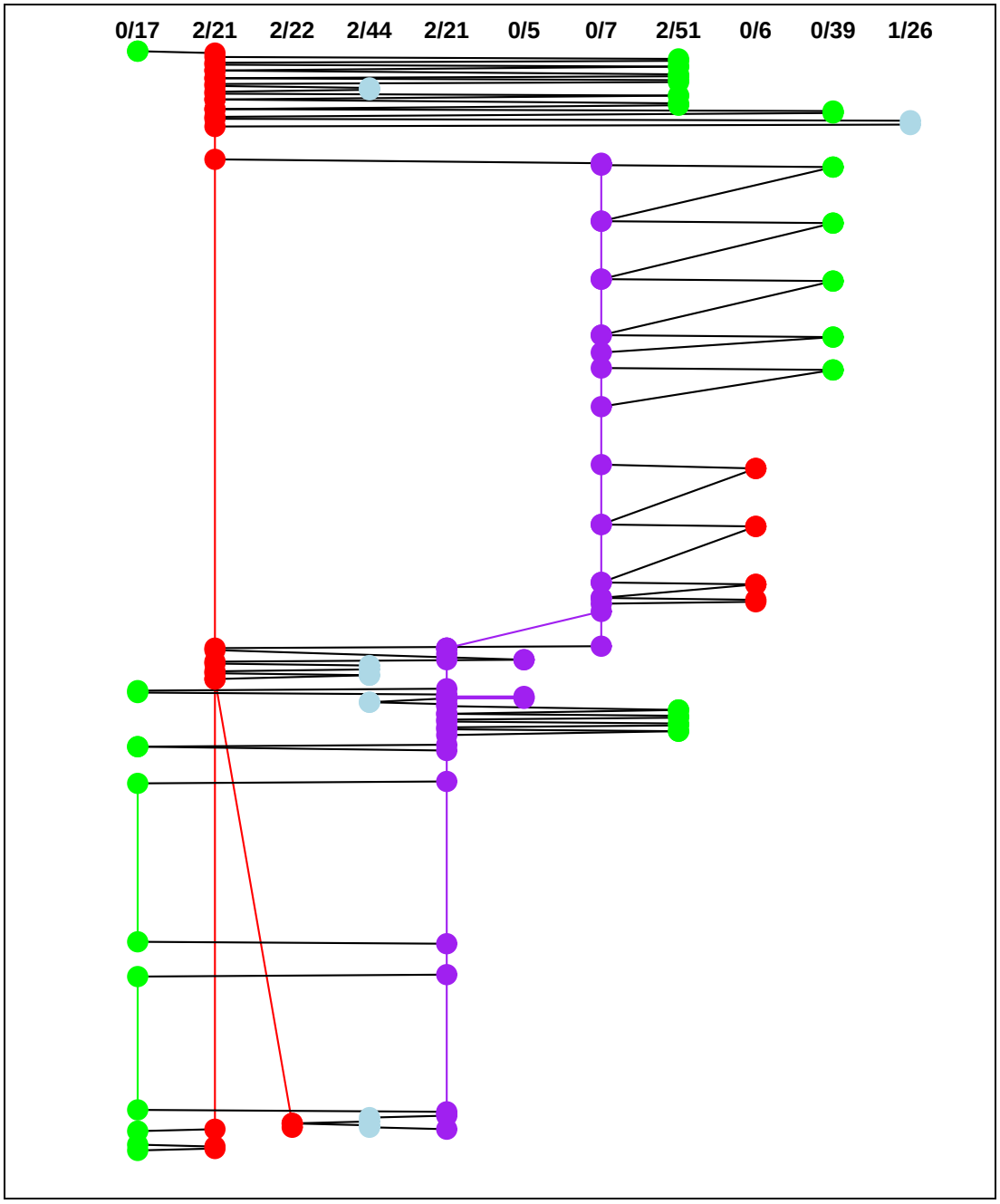

Figure 2: Interaction network showing system reaction to a newline input to the Bourne shell to request execution of `ls`.

thread id	machine	description
0/17	scooter	tty server
2/21	ralph	shell (main thread)
2/22	ralph	shell (waiting thread)
2/44	piggy	session server
2/21	grover	ls
0/5	grover	process/segment server
0/7	grover	process/segment server
2/51	scooter	soap (directory server)
0/6	ralph	process/segment server
0/39	scooter	bullet (file server)
1/26	piggy	run server

Table 1: Key to the threads shown in Figure 2.

is executing). This segment probably contains startup information, such as the environment for the new process.

With the three segments loaded the new thread is created, with the first activity in the thread represented by the diagonal edge from thread 0/7 to thread 2/21. This completes the RPC to create the `ls` process, so the process/segment server replies to the shell.

The shell then updates information held by the session server, before creating a thread to wait for `ls` to terminate. `ls` consults the directory server to extract the information it needs, then writes the directory listing to the terminal server in two large writes, of 79 and 66 bytes respectively. The terminal server takes a considerable time to reply to each of these calls. In each case the delay reflects the time taken to write the bytes supplied at a rate of 9600 baud.

Just before the `ls` thread terminates, it makes a call to the session server to inform it that `ls` is about to exit. The session server passes this information on to the shell thread created to wait for `ls` to terminate. This shell thread then wakes up the main shell thread before terminating. The main shell thread makes an RPC to the terminal server to print the prompt for the next command line, before sending a request to the terminal server for further input.

Various points of interest that can be seen from this network include:

1. Much of the interaction network is characteristic of commands run from the Bourne shell. The processing up to the creation of `ls`, including access to the directory server, and the whole RPC to create the new process, is a standard pattern of communication. The size of the executable loaded determines the number of accesses to the file server, but the overall pattern is the same. Also, the much smaller bottom part of the network from where `ls` makes its final RPC to the session server is characteristic end-of-command processing.

2. If the `ls` executable is already cached on the machine selected to run it, then the file server accesses to load the executable do not occur.

3. Of the 11 threads, 9 existed before the user action occurred. The other two were created (and terminated) during the system reaction. One of these threads was created in an

existing process (the Bourne shell) to wait for the `ls` process to finish. The other was created in a new address space to run the `ls` program.

Note the two different methods of thread creation shown in Figure 2. The Bourne shell main thread creates a new thread in the same address space directly, by making a system call. The Bourne shell main thread creates a new thread in a new address space in an indirect fashion by making a call to the process/segment server on the machine on which `ls` is to be run.

4. Vertices representing FLIP send and FLIP receive events are not shown in the interaction network, as they tend to clutter the display. Several long reply messages from the file server and from the process/segment server on ralph can be seen in the upper half of the network. These 30000 byte messages are represented by edges with a long elapsed time. Each 30000 byte packet at the RPC level is fragmented into 21 ethernet packets at the FLIP level.

5. The Bourne shell has been optimised for use with Amoeba, in that a new copy of the shell is not forked off whenever a new command is executed. Figure 3 shows what happens when `ls` is run from the Korn shell, a shell that has not been optimised in this way. In the figure, it can be seen that two new processes (as opposed to threads) are created. The first process created (process 3 on ralph) is a new copy of the Korn shell. Two threads are created in process 31. The first (20) waits for POSIX signals to occur. The second (31) is the command interpreter, and it is the thread that creates the `ls` process (thread 4/33 on piggy). The `ls` thread itself occupies a relatively small part of the network.

   The benefits gained by optimising the Bourne shell to avoid creating a new copy of the shell are obvious from a comparison of the two networks. The response time for `ls` executed from the Korn shell was 1.009 seconds, as against 0.569 seconds when `ls` was executed from the Bourne shell.

7 Case 3—`elvis`

Amoeba is used in an operating systems course in the School of Electrical Engineering at University of Technology, Sydney. In discussions with David Holmes from UTS, it became apparent that they had been having performance problems with `elvis`, a version of the `vi` editor distributed with Amoeba. Some interaction networks that resulted from inputs to `elvis` were subsequently recorded, and two of those interaction networks are presented in this section.

7.1 Insertion of a single character

The first interaction network resulted from a keystroke made during input of characters into the buffer being edited. The interaction network recorded is shown in Figure 4. Three threads from three processes on two machines are involved, as summarised in Table 2.

The terminal server sends a reply to an earlier RPC made by `elvis` to return the character input. `elvis` then makes an RPC to the session server, before making an RPC to the terminal

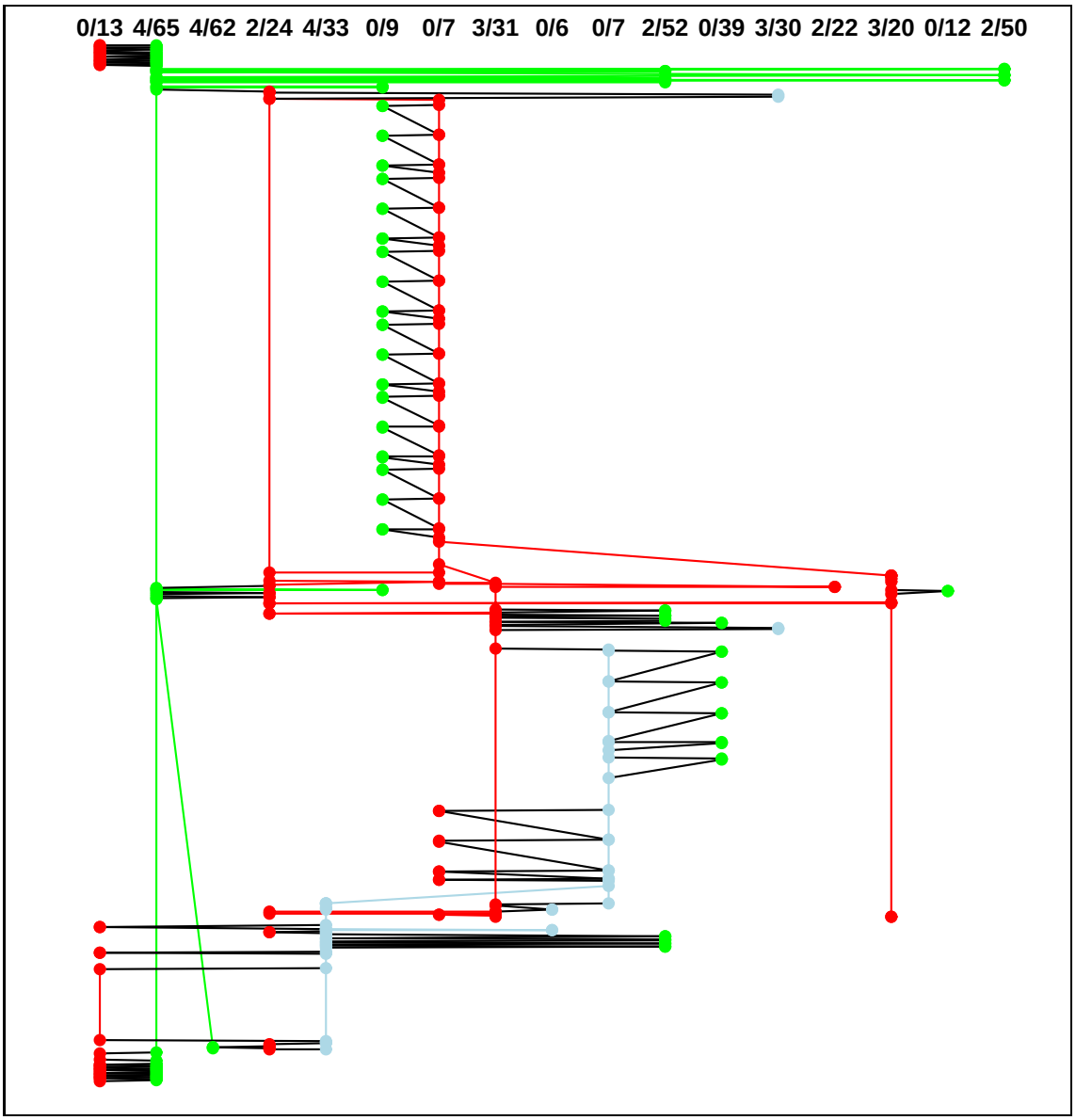

Figure 3: Interaction network showing system reaction to a newline input to the Korn shell to request execution of `ls`.

thread id	machine	description
0/18	scooter	tty server
2/26	grover	elvis
4/59	scooter	session server

Table 2: Key to the threads shown in Figure 4.

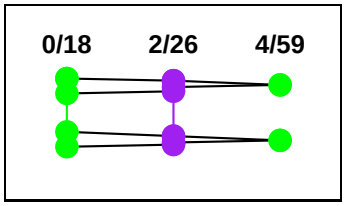

Figure 4: Interaction network showing system reaction to a single character inserted into an `elvis` buffer.

server to perform output. After that, `elvis` makes a second RPC to the session server, before sending a request to the terminal server to ask for more input.

Nineteen characters are written by `elvis`, and this is reflected by the length of the edge representing the bytes being output by the terminal server. The reason for this many characters being written, rather than only the single character just input, has not been determined.

The message passing shown in Figure 4 can be compared with message passing triggered by input of characters on a shell command line. For the Bourne shell, two messages are involved. The first is sent by the terminal server, and is a reply containing the character input. The shell then sends a message to the terminal server asking for the next character. Character echoing is performed by the terminal server. For the Korn shell, four messages are exchanged. The same two messages as for the Bourne shell, plus two messages for an RPC from the shell to the terminal server to echo the character (the Korn shell provides command line editing, so must do the echoing itself).

The response time for the interaction network shown in Figure 4 is 32.2ms, of which about 18ms is taken up by writing of 19 characters to the terminal. Four RPCs were needed, requiring eight ethernet packets. To improve performance, reasons for the two RPCs made to the session server could be investigated to see whether they can be eliminated. Also, the run server could be changed to consider communication patterns as well as machine load when process placement decisions are made.

7.2 Input of colon

The second `elvis` interaction network shows in Figure 5 the reaction to input of a colon character. The input of the colon precedes input of an `elvis` command line. Six threads from five processes on two machines are involved, as summarised in Table 3.

The message passing in Figure 4 is also evident at the top and the bottom of Figure 5. In between, `elvis` makes several RPCs to the file and directory servers, with the directory server making RPCs to the file and disk servers. The elapsed times of the RPC to the disk server is both 23ms, and is most likely due to disk access delays. Reasons have yet to be established for the lengthy delay of 35ms in one of the RPCs made to the file server.

The response time for this interaction network is 182ms. A total of 14 RPCs, and 24 ethernet packets are needed. Performance-wise, this seems to be an awful lot of work in response to a character that merely prefixes an `elvis` command.

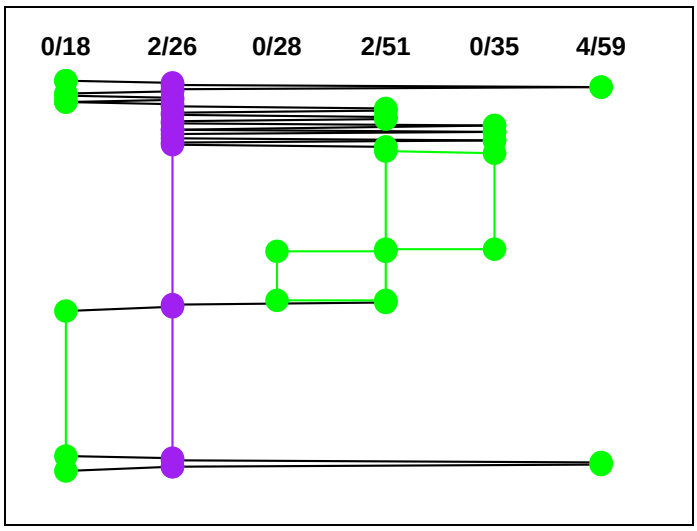

Figure 5: Interaction network showing system reaction to input to `elvis` of a colon character that prefixed a command.

thread id	machine	description
0/18	scooter	tty server
2/26	grover	elvis
0/28	scooter	disk server
2/51	scooter	soap (directory server)
0/35	scooter	bullet (file server)
4/59	scooter	session server

Table 3: Key to the threads shown in Figure 5.

thread id	machine	description
0/18	scooter	tty server
2/24	ralph	shell (main thread)
2/20	ralph	shell (waiting thread)
2/34	piggy	session server
4/22	ralph	aps
0/6	ralph	process/segment server
0/2	gonzo	dir server
2/51	scooter	soap (directory server)
0/6	grover	process/segment server
0/2	grover	dir server
0/7	piggy	process/segment server
0/2	piggy	dir server
0/2	ralph	dir server
0/9	scooter	process/segment server
0/2	scooter	dir server
0/39	scooter	bullet (file server)
1/26	piggy	run server

Table 4: Key to the threads shown in Figure 6.

8 Case 4—aps

The interaction network in Figure 6 shows execution of the `aps` (Amoeba `ps`) command initiated from the Bourne shell. `aps` in this instance was run with the options `-a -m`, which causes all processes on all machines to be listed. Seventeen threads running in 10 processes spread over five machines are involved in performing the system reaction to the entry of newline at the end of the `aps` command line. The threads involved are summarised in the Table 4. The dir server on each machine provides directory information for local capabilities, including capabilities for local devices and processes. The soap directory server provides a general directory service, which provides access to the capabilities of all of the per-host directory services.

The message passing patterns leading up the the creation of the `aps` process, and that occur when `aps` terminates are standard for commands run from the Bourne shell, and have already been discussed in Case 2 above. `aps` starts by writing a heading to the terminal server. It then enters a loop in which it extracts information from all processes on one machine. The loop starts with an RPC to the soap server, probably to get the capability needed to access the dir server on the machine in question. The dir server on that machine is then contacted to determine the number of user-mode processes on that machine. An inner loop is then entered to get per-process information. Each time around the loop the dir server and process/segment server are consulted to retrieve per-process information. Finally in the inner loop the information obtained is written to the terminal server.

The response time for this interaction network is 983ms. A total of 75 RPCs, and 185 ethernet packets were needed.

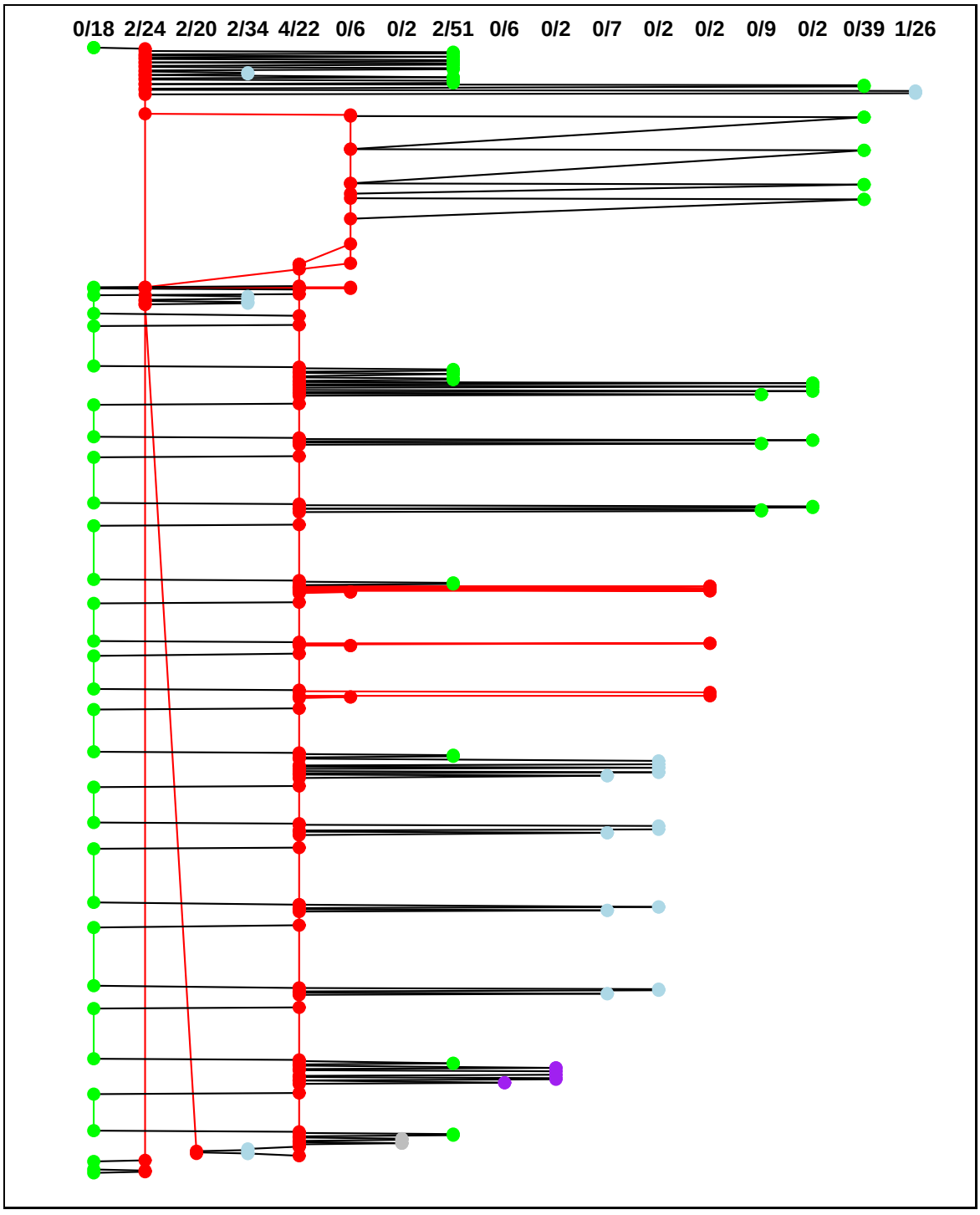

Figure 6: Interaction network showing system reaction to a newline input to the Bourne shell to request execution of `aps`.

thread id	machine	description
0/18	scooter	tty server
2/24	ralph	shell (main thread)
2/22	ralph	shell (waiting thread)
2/51	scooter	soap (directory server)
2/39	piggy	session server
2/34	piggy	session server
2/45	piggy	session server
2/48	piggy	session server
4/22	ralph	dir
0/6	ralph	process/segment server
0/39	scooter	bullet (file server)
0/5	gonzo	process/segment server
0/7	gonzo	process/segment server
2/20	gonzo	wc
1/26	piggy	run server

Table 5: Key to the threads shown in Figure 7.

9 Case 5—A simple process pipeline

The interaction network in Figure 7 shows execution of the `dir | wc` command line input to the Bourne shell. Fifteen threads running in nine processes spread over four machines are involved in performing the system reaction to the entry of newline at the end of the command line. The threads involved are summarised in Table 5.

After reading the command line, the shell makes an RPC to the session server to create the pipe. This causes the session server to create two threads (2/45 and 2/48 on piggy) to service RPCs related to the new pipe. As it happens, thread 2/45 handles all of the RPCs made to the pipe server, with thread 2/48 simply exiting after receiving an RPC telling it to.

The first process in the pipeline, `dir`, is created on ralph. Thread 4/22 executes the `dir` program. Because the process is created on the same machine as the shell, no RPCs are needed in loading of the third segment (compare this with what happened in Case 2). `dir` writes its output to the pipe in a single RPC to thread 2/35, then exits shortly after.

Concurrent with the execution of `dir`, the shell initiates creation of `wc`, which is executed by thread 2/20 on gonzo. `wc` reads from the pipe via an RPC to thread 2/45, then writes its output to the terminal server. When `wc` closes the pipe the two pipe server threads exit.

An interesting feature of this network is that two messages (both sent to thread 2/45) have very long delays. One message, with a delay of about 300ms, was sent by thread 4/22. The other message, with a delay of 126ms, was sent by thread 2/20. Examination of the textual dump of event record data showed that in both cases the delay occurred between the send event at the RPC layer and the send event at the FLIP layer. Further investigation is needed to determine the exact cause of these lengthy delays.

Another long delay occurs in file server thread 0/39 while it is servicing an RPC made by the shell. It takes 85ms to service the call. This long delay is similar to a long delay in file server response observed in Figure 5. If the file server is able to access the disk directly

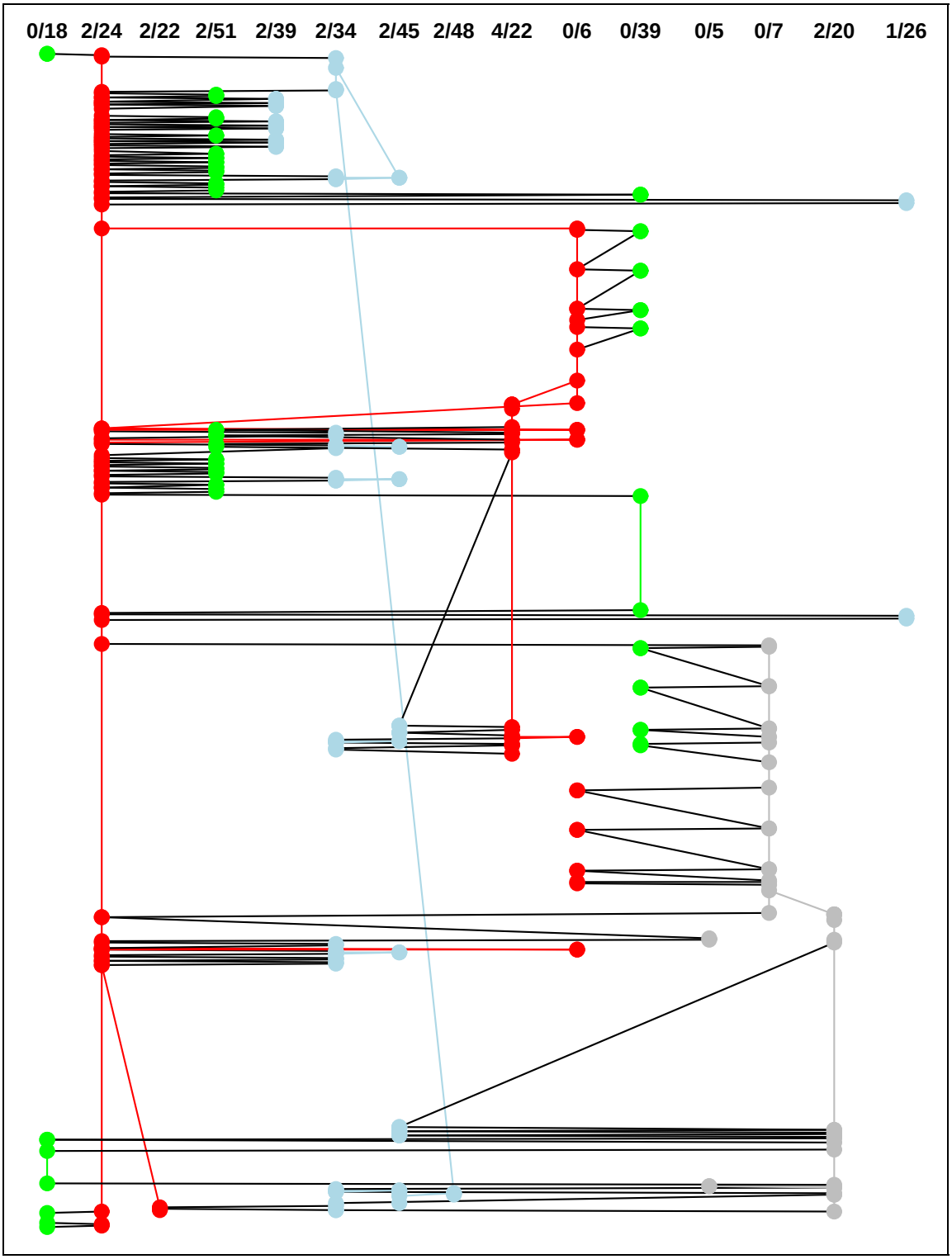

Figure 7: Interaction network showing system reaction to a newline input to the Bourne shell to request execution of `dir | wc`.

(rather than going through the disk server) then this might account for the long delays. Further investigation is needed to determine the reasons for this delay.

The response time for this interaction network is 875ms. A total of 82 RPCs, and 287 ethernet packets were needed.

10 Experiences to date

Based on the limited experience we have had with the monitor to date, we can report that:

- Although SunOS and Amoeba are very different operating systems, the design of the SunOS monitor has been directly applicable to the Amoeba monitor. In fact this has also been the case for much of the SunOS monitor's source code particularly, although not exclusively, the source code for the analysis tools.

- Propagation of interaction identifiers through remote procedure calls has been sufficient to allow recording of all processing for a wide variety of commands. Additional code was required to correctly propagate the interaction identifier through the `fork` call in the Unix emulation library. During the fork call, an RPC to the session server is triggered in a rather indirect fashion, that results in the RPC appearing to originate from a shell thread not involved in the interaction. The code added ensures that the correct interaction identifier accompanies the RPC request message that is eventually sent.

  The Amoeba group message passing primitives have yet to be instrumented. So far this has not caused any problems, as none of the programs monitored have made use of these primitives. Also, several Amoeba synchronisation primitives, such as the mutex and the signal, have yet to be instrumented. This is apparent from Figure 2. At the bottom of the figure, the waiting shell thread (2/22) awakens the main shell thread (2/21) by unlocking a mutex. There is no edge in the interaction network, however, that reflects this piece of communication.

- The interaction networks recorded to date have given considerable insights into the inner workings of Amoeba. A bug in the MMU context flushing code for the sun4c architecture was detected using the monitor. Interaction networks showing system reactions for inputs to `elvis` have revealed substantial reactions for inputs that should be handled very quickly. Further investigation is needed to determine possible tuning actions.

  Some messages have been recorded that have very long delays between the send events at the RPC and FLIP protocol levels. This needs to be investigated to determine whether these delays should be occurring or not.

11 Conclusions

The following conclusions are based on the preliminary experiences with the Amoeba interaction network monitor:

- The interaction network concept, and the design and much of the source code of the SunOS version, have been very easily applied to Amoeba. The process of deciding on the location of probe points in a new kernel is never easy, but knowing where probes had been placed in the SunOS kernel helped during addition of the Amoeba probes.

- The interaction networks recorded so far have given interesting insights into Amoeba's operation, and have lead to the identification of at least one bug.
- The value of interaction monitoring (as against program monitoring) is clear from these examples. A lot of processing occurs in servers that are invoked directly or indirectly from the shell and the program(s) being run.
- A systematic study of Amoeba performance has not been undertaken. The amount of ethernet traffic evident from the interaction networks reported here shows that the network is a likely bottleneck. The interaction network monitor is a valuable tool in determining the causes of network traffic generated by a user action.

References

[1] Paul Ashton. *The Interaction Network: a Performance Measurement and Evaluation Tool for Loosely-Coupled Distributed Systems.* PhD thesis, Department of Computer Science, University of Canterbury, Christchurch, New Zealand, March 1992.

[2] Paul Ashton and John Penny. Monitoring the processing of interactive requests on distributed systems. Technical Report TR-COSC07/95, University of Canterbury, Department of Computer Science, October 1995. URL: `http://www.cosc.canterbury.ac.nz/~paul/tr-cosc.07.95.ps.gz`.

[3] Paul Ashton and John Penny. A tool for visualising the execution of interactions on a loosely-coupled distributed system. *Software—Practice and Experience*, (accepted for publication).

[4] A. Duda, G. Harrus, Y. Haddad, and G. Bernard. Estimating global time in distributed systems. In *7th International Conference on Distributed Computing Systems*, pages 299–306. IEEE, September 1987.

[5] S. J. Mullender, G. van Rossum, A. S. Tanenbaum, R. van Renesse, and H. van Staveren. Amoeba: A distributed operating system for the 1990s. *Computer*, 23(5):44–53, May 1990.

[6] A. S. Tanenbaum, R. van Renesse, H. van Staveren, G. J. Sharp, S. J. Mullender, J. Jansen, and G. van Rossum. Experiences with the Amoeba distributed operating system. *Communications of the ACM*, 33(12):46–63, December 1990.